REMARKS

The present amendment is prepared in accordance with the new revised requirements of 37 C.F.R. § 1.121. A complete listing of all the claims in the application is shown above showing the status of each claim. For current amendments, inserted material is underlined and deleted material has a line therethrough.

Applicants appreciate the thoroughness with which the Examiner has examined the above-identified application. Reconsideration is requested in view of the amendments above and the remarks below.

Claim 23 has been canceled.

New claim 26 has been added. Claim 26 includes the limitations of base claim 16 and dependent claim 23.

No new matter has been added.

Claim Rejections - 35 USC § 102

The Examiner has rejected claims 1-7, 9-11, 14-19, and 23-25 under 35 USC 102(b) as being anticipated by The GNU Make Manual (GNU MM), Version 3.79, edition 0.55, (04/04/2000). Applicant disagrees.

Applicant continues to submit that the present invention is aimed at improving upon conventional *make* utilities by providing a software utility program for automating selective updates of source code and corresponding object code. (Specification, page 1, lines 5-6.) In so doing, independent claim 1 is directed to a method for updating existing code in a computer program after inputting new code that defines changes to the existing code. The method includes generating a target file list of target files to update, and a dependency file list of files dependent on the target files. The dependency file list is then read into a control file, and once therein, selected lines of these files are split into target strings and prerequisite strings. Those target strings having programming language substitutions are appended to a requisition list, while the prerequisite strings are stored in corresponding requisite arrays thereto. An algorithm is executed, whereby this algorithm first matches target files from the target file list with the substituted target string in the requisition list in the control file. The algorithm then updates those matched target files from the target file list if it is determined that the corresponding prerequisite strings, which are stored in the corresponding requisite arrays in the control file, have been updated more recently than the substituted target string.

Similarly, independent claim 5 is directed to a method for generating changes and updating existing files and code in a computer program by first reading existing target files and dependency files in the computer program. A plurality of these dependency files associated with the target files are then read into a single control file. Once in the single control file, selected lines of the dependency files are split into target strings and prerequisite strings. A utility program is executed for updating the target files and associated dependency files, whereby the utility program includes an interpreted scripting language specifying particular characters to search for in the target files and associated dependency files. Using the utility program, a requisition list of target strings having interpreted scripting language substitutions is then generated, as well as corresponding requisite arrays for the prerequisite strings. The target files are then updated by employing a search technique defined in the utility program that includes specified target patterns identifying the existing target files being updated. The existing target files are updated if it is determined that the prerequisite strings in the control file have been updated more recently than the substituted target string.

Further, independent claim 16 is directed to a method for updating target files in a computer by generating a target file list of target files to update and reading <u>into a control file</u> a list of files dependent on the target files. Selected lines of these dependent files are split into target strings and prerequisite strings, and then programming language substitutions are performed in the target strings. The substituted target strings are appended to a requisition list, which is stored in corresponding requisite arrays. An algorithm is then executed to match selected target files from the target file list to the substituted target string in the requisition list. The prerequisite strings are then retrieved from the corresponding requisite arrays and updated by performing all possible programming language substitutions using the algorithm. Using the algorithm, those prerequisite strings that have been updated more recently than the substituted target string are identified to generate update rules, and then the target files from the target file list are updated using such update rules.

Applicant submits that The GNU Make Manual is directed to yet another prior art program that uses a conventional "*make*" utility, at which the present invention is aimed at improving upon and overcoming the problems associated therewith. (See, Specification, p. 1, l. 8 – p. 4, l. 25.) Again, as recited in applicant's application, the present invention provides improvements to conventional *make* utilities that use descripter files containing dependency rules, macros, and suffix rules to instruct *make* to automatically rebuild the program whenever one of the program component files is modified. The *make* utility operates by following rules that are provided in its descriptor file, typically called *makefile*. (Specification, p. 1, ll. 20-23.)

As disclosed in applicant's application, a conventional *makefile* includes two elements: targets, and dependencies, whereby the *make* utility compares the relationships therebetween, particularly the time stamp of each target and its dependencies, i.e., *make* operates by comparing the date and time of a source file with the date and time of the associated object file. (Specification, p. 2, ll. 1-6 and p. 3, ll. 28-29.) If the comparison shows that the source file is newer then the object file, or if the object file does not exist, *make* performs the task listed in the *makefile* to convert the source file into an object file. In contrast, if the object file is newer then the source file, then *make* recognizes that is does not have to recompile the source file. (Specification, p. 3, ll. 1-5.) Multiple *makefiles* are needed to specify cross-directory file dependencies. (Specification, p. 3, ll. 12-13.) The *make* utility requires explicitly detailing the source files or target files needing to be changed or updated which can take significant amounts of time and be cumbersome. Also, the *make* command requires building a substantially complete dependency tree before it starts, which can also take an extensive amount of time and computer resources. (Specification, p. 2, ll. 13-17.)

The GNU Make Manual discloses a *make* utility that automatically determines which pieces of a large program needs to be recompiled, and issues the commands to recompile them. (GNU MM, p. 1, ll. 23-25 and p. 6., ll. 3-5.) To prepare to use *make*, one must write a *makefile* that describes the relationships among files in the program and provides commands for updating each file. (GNU MM, p. 6., ll. 20-22.) Once a suitable *makefile* exists, each time a source file is changed, a simple shell command performs recompilations. (GNU MM, p. 6., ll. 26-28.) The *make* program uses the *makefile* data base and the last modification times of the fields to decide which of the files need to be updated, and then issues the command recorded in the data base. (GNU MM, p. 6., ll. 28-31.)

A *makefile* contains five kinds of things: explicit rules, implicit rules, variable definitions, directives, and comments. (GNU MM, p. 15, l. 28-30.) The GNU *make* does its work in two distinct phases. (GNU MM, p.21, l. 41.) During the first phase, *make* reads all the *makefiles*, and internalizes all the variables and their values, implicit and explicit rules, and constructs a dependency graph of all the targets and their prerequisites. (GNU MM, p.21, ll. 41-45.) During the second phase, *make* uses these internal structures to determine what targets will need to be rebuilt and to invoke the rules necessary to do so. (GNU MM, p.21, l. 45-47.) When the objects of a *makefile* are created only by implicit rules, entries may be grouped by their prerequisites instead of by their targets. (GNU MM, p.14, ll. 9-11.) The commands of a rule consist of <u>shell command</u> lines to be executed one by one. (GNU MM, p. 47, ll. 35-36.) Users use many different shell programs, but commands in *makefiles* are always interpreted by /bin/sh unless the *makefile* specifies otherwise. (GNU MM, p. 47, ll. 43-45.)

Page 9, lines 35-50 of The GNU Make Manual show a straightforward *makefile* that describes the way an executable file called *edit* depends on eight object files, which in turn depend on eight C source and three header files. (GNU MM, p. 9, ll. 29-31.) In the example *makefile*, the targets include the executable file *edit*, and the object files "*main.o*" and "*kbd.o*", whereby each ".o" file is both a target and a prerequisite. When a target is a file, it needs to be recompiled or relinked if any of its prerequisites change. In addition, any prerequisites that are themselves automatically generated should be updated first. (GNU MM, p. 10, ll. 21-28.) A shell command follows each line that contains a target and prerequisites, whereby these shell commands say how to update the target file. (GNU MM, p. 10, ll. 33-35.) A tab character must come at the beginning of every command line to distinguish commands lines from other lines in the makefile. (Bear in mind that *make* does not know anything about how the commands work. It is up to you to supply commands that will update the target file properly. All *make* does is execute the commands in the rule you have specified when the target file needs to be updated.) (GNU MM, p. 10, ll. 35-40.)

Applicant submits that the present invention is not anticipated by The GNU Make Manual. Anticipation is but the ultimate or epitome of obviousness. To constitute anticipation, all material elements of a claim must be found in one prior art source. In re Marshall, 577 F.2d 301, 198 USPQ 344 (CCPA 1978).

With respect to the Examiner's arguments, The GNU Make Manual does not disclose reading a plurality of dependency files associated with target files into a single control file and splitting such dependency files into target strings and prerequisite strings. Rather, the pages of The GNU Make Manual cited by the Examiner (GNU MM, p.14, ll. 9-11) merely disclose that "[w]hen the objects of a *makefile* are created only by implicit rules, entries may be grouped by their prerequisites instead of by their targets. The GNU Make Manual does not disclose or even contemplate reading dependency files into a control file or single control file, and once therein, splitting selected lines of these dependency files into target strings and prerequisite strings, as is currently claimed.

Also, The GNU Make Manual does not recite performing programming language substitutions in these target strings using an interpreted scripting language, whereby target strings having programming language substitutions are appended to a requisition list while the prerequisite strings are stored in corresponding requisite arrays thereto. (See, claims 1, 5 and 16.) The GNU Make Manual merely discloses that the commands of a rule consist of shell command lines to be executed one by one. (GNU MM, p. 47, ll. 35-36.) Users use many different shell programs, but commands in *makefiles* are always interpreted by /bin/sh unless the *makefile* specifies otherwise. (GNU MM, p. 47, ll. 43-45.) This is not an interpreted scripting language. As one skilled in the art will understand, there are two parts to an update rule: the conditional (when something needs to be done) and the action (what might need to be done). The "interpretation" referred to in The GNU Make Manual refers to the interpretation of the shell commands which are part of the "'action". Applicant's claimed "interpreted scripting language" is part of the system for processing the 'conditional' part of the rules. This distinction is central to the present invention because the use of interpretation in processing the rules themselves provides critical support for multi-directory builds from a single control file, and is considerably more universal and powerful than simple textual substitution.

As The GNU Make Manual does not disclose reading dependency files into a control file (i.e., single control file) it cannot and does not disclose updating the split target files if it is determined that the prerequisite strings <u>in the control file</u> have been updated more recently than the substituted target string (claim 5). The GNU Make Manual does not disclose performing such a step by executing an algorithm that <u>matches</u> target files from the target file list with the substituted target string in the requisition list <u>in the control file</u> (claims 1 and 16) if it is determined that the corresponding prerequisite strings, which are stored in and retrieved from the corresponding requisite arrays in the control file, have been updated more recently than the substituted target string (claims 1 and 16). That is, those prerequisite strings that have been updated more recently than the substituted target strings are identified to generate update rules, and then the target files from the target file list are updated using such update rules (claim 16). Amended independent claims 14 and 15, which are respectively directed to a computer program product and a program storage device for executing the method of amended claim 1, also include the above limitations that distinguish amended claim 1 from the cited GNU Make Manual.

The Examiner cites page 9, lines 35-50, of The GNU Make Manual, which merely shows a straightforward *makefile*. (GNU MM, p. 9, ll. 29-31.) As dislcosed in The GNU Make Manual, a shell command follows each line that contains a target and prerequisites, whereby these shell commands say how to update the target file. (GNU MM, p. 10, ll. 33-35); however, "bear in mind that *make* does not know anything about how the commands work. It is up to you to supply commands that will update the target file properly. All *make* does is execute the commands in the rule you have specified when the target file needs to be updated" (GNU MM, p. 10, ll. 35-40).

Again, the present invention provides improvements to conventional *make* utilities that use descripter files, typically called *makefiles,* containing dependency rules, macros, and suffix rules to instruct *make* to automatically rebuild the program whenever one of the program component files is modified. (Specification, p. 1, ll. 20-23.) In so doing, a critical feature of applicant's invention is that a plurality of dependency files associated with target files are read <u>into a control file</u> (a single control file), and split into target and prerequisite strings, whereby those target strings having programming language substitutions are appended to a requisition list while the prerequisite strings are appended to corresponding requisite arrays. Using an algorithm that includes specified target patterns, the target files are updated if it is determined that the prerequisite strings in the control file have been updated more recently than the substituted target string. The GNU Make Manual does not disclose any of the foregoing.

Moreover, with respect to new independent claim 26, which includes the limitations of base claim 23 and dependent claim 16, new claim 26 recites that update rules are generated using the algorithm by identifying those prerequisite strings that have been updated more recently than the substituted target string. These update rules support multi-directory builds from a single control file. Applicant submits that The GNU Make Manual's Gmake's vpath/VPATH facility does not disclose multi-directory builds from a single control file. It is a conventional *make* utility that uses conventional *makefiles*, whereby multiple *makefiles* are needed to specify cross-directory file dependencies. (Specification, p. 2, l. 6 to p. 3 l. 13.) Further, the passages of The GNU Make Manual cited by the Examiner, namely, "The directory search features of make facilitate this by <u>searching several directories</u> automatically to find a prerequisite" at page 28, lines 51-53, do not disclose update rules that support multi-directory <u>builds</u> from a single control file.

Moreover, it is respectfully submitted that one skilled in the art will appreciate that The GNU Make Manual's *gmake* vpath/VPATH has no way of processing multiple source files having the same name but which are located in different source subdirectories, unless multiple *gmake* control files are used. Additionally, consider the problem of using *gmake* to copy all recently changed files in a source directory tree SRCDIR to a target directory tree TRGDIR. Using *gmake*, one would have to write a separate control file for each subdirectory in the SRCDIR subtree, because vpath/VPATH is inadequate for this application, and cannot be extended to handle it. The present invention enables one to generate update rules that support multi-directory builds from a single control file, as is currently claimed.

Also, the "directory searching" in The GNU Make Manual's *gmake* is not dynamic directory switching to specify multiple files in multiple directories as is recited in claim 24. Applicant's dynamic directory switching allows the user to conveniently switch from one directory to another as he or she specifies prerequisites. (See, specification, p. 14, l. 26 to p. 15, l. 7.) With respect to claim 25, The GNU Make Manual does not disclose that an algorithm considers a directory to be <u>out-of-date regardless of its time stamp</u> (i.e., always out-of-date regardless of its time stamp) such that any rule associated with directory target is always triggered. The pages cited by the Examiner, p. 120 line 5 - p. 127 line 50 "(pattern matching rules and automatic variables are used to specify rules for directories)" do not support the recited limitations of claim 25.

It is for the above reasons that applicant submits that independent claim 1 and claims 2-4 dependent thereon, independent claim 5 and claims 6, 7, and 9-11 dependent thereon, independent claims 14 and 15, independent claim 16 and claims 17-22 and 24-25 dependent thereon, and new independent claim 26 are not anticipated by The GNU Make Manual.

Applicant submits that the claims of the instant invention include limitations not disclosed nor contemplated by The GNU Make Manual such that The GNU Make Manual does not anticipate nor render obvious the instant invention.

<u>Claim Rejections - 35 USC § 103</u>

Claims 12, 13, 20-22 are rejected under 35 USC 103(a) as being unpatentable over The GNU Make Manual, Version 3.79, edition 0.55, (04/04/2000) in view of Welch, "Practical Programming in Tcl and Tk", (1999).

Applicant continues to submit that The GNU Make Manual does not disclose, contemplate or suggest reading a plurality of dependency files associated with target files into a single control file (i.e., a control file) and splitting such dependency files into target strings and prerequisite strings (claims 5 and 16). As such, it does not disclose, contemplate or suggest performing programming language substitutions in these target strings using an interpreted scripting language (claim 5), including *updt*, perl, or Tcl (claim 12), whereby target strings having programming language substitutions are appended to a requisition list while the prerequisite strings are stored in corresponding requisite arrays thereto (claims 5 and 16). Also, as The GNU Make Manual does not disclose reading dependency files into a control file (i.e., single control file) it cannot, and does not disclose, contemplate or suggest executing a utility program to update the split target files if it is determined that the prerequisite strings in the control file have been updated more recently than the substituted target string (claims 5 and 16). The utility program includes an interpreted scripting language specifying particular characters to search for in the target and associated dependency files (claim 5), whereby existing target files with a specific character are not considered to be a file, and thereby are bypassed for any changes by the utility program (claim 13).

Further, The GNU Make Manual does not disclose performing such a step by executing an algorithm that matches target files from the target file list with the substituted target string in the requisition list (claim 16) in the control file if it is determined that the corresponding prerequisite strings, which are stored in and retrieved from the corresponding requisite arrays in the control file, have been updated more recently than the substituted target string (claim 16). That is, those prerequisite strings that have been updated more recently than the substituted target strings are identified to generate update rules, and then the target files from the target file list are updated using such update rules (claim 16). The update rules are specified using a scripting language selected from *updt*, perl, and Tcl (claim 20), whereby the algorithm is executed in such scripting language (claim 21) and the rules have access to the interpreted programming language to recursively invoke the algorithm on a new target (claim 22).

The Examiner has cited Welch, "Practical Programming in Tcl and Tk", (1999), however, applicant submits that the Welch reference does not overcome the above deficiencies of The GNU Make Manual. Welch is merely directed Tcl scripting and is delivered as a practical guide to help users of the Tcl and Tk programming languages get the most out of Tcl and Tk. (Welch, p. xivi, ll. 5-9 and 17.) It merely sets up a set of programming techniques that exploit the power of Tcl and Tk while avoiding troublesome areas. (Welch, p. xivi, ll. 7-8.) The Welch reference does not disclose, contemplate or suggest reading a plurality of dependency files associated with target files into a control file (i.e., a single control file) and splitting such dependency files into target strings and prerequisite strings, as is currently claimed. As such, Welch does not disclose, contemplate or suggest performing programming language substitutions in these target strings using an interpreted scripting language, and executing a utility program to update the split target files if it is determined that the prerequisite strings in the control file have been updated more recently than the substituted target string, also as is claimed. Welch does not overcome the above deficiencies of The GNU Make Manual.

Applicant submits that neither The GNU Make Manual nor the Welch reference, taken singly or in any proper combination thereof, discloses the instant invention, such that, pending claims 12, 13 and 20-22 are neither anticipated by nor rendered obvious over The GNU Make Manual or the Welch reference, alone or in any proper combination. It is submitted that claims 12, 13 and 20-22 are properly allowable for the reasons set forth above.

It is respectfully submitted that the application has now been brought into a condition where allowance of the case is proper. Reconsideration and issuance of a Notice of Allowance are respectfully solicited. Should the Examiner not find the claims to be allowable, Applicants' attorney respectfully requests that the Examiner call the undersigned to clarify any issue and/or to place the case in condition for allowance.

Respectfully submitted,

[illegible]

Kelly M. Reynolds
Reg. No. 47,898

DeLIO & PETERSON, LLC
121 Whitney Avenue
New Haven, CT 06510-1241
(203) 787-0595

CERTIFICATE OF MAILING

I hereby certify that this correspondence is being deposited with the United States Postal Service on the date indicated below as first class mail in an envelope addressed to Mail Stop AF, Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450.

Name: Carol M. Thomas Date: July 14, 2004 Signature: [illegible]

ibmf100320000amdB